Efficient programming with R: Some tips to reduce time in computer intensive programs

Josep A.Sanchez-Espigares

Dept. Statistics and Operational Research
UPC- BarcelonaTech

Jornades de Consultoria Estadística i Software - 2011

CosmoCaixa Barcelona, Setember 26-28th, 2011

Outline

- Motivation
- Programming with R
  - use fast instructions
- Adding Compiled Code
  - avoid interpretation for recurrent processes

Outline

- Motivation
- Programming with R
  - use fast instructions
- Adding Compiled Code
  - avoid interpretation for recurrent processes
- Taking full advantage of hardware
  - parallelization and GPUs

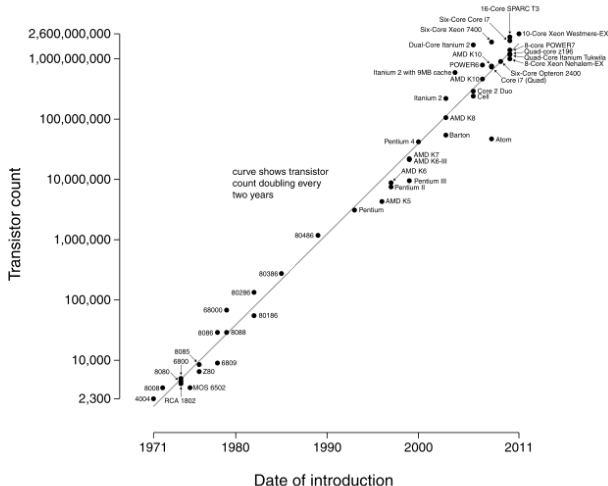
Outline

- Motivation
- Programming with R
  - use fast instructions
- Adding Compiled Code
  - avoid interpretation for recurrent processes
- Taking full advantage of hardware
  - parallelization and GPUs
- Conclusions

Outline

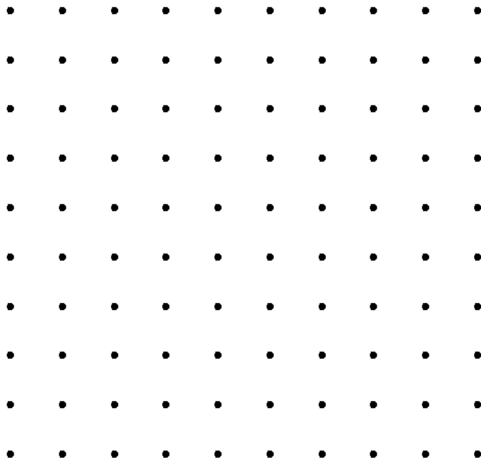
- Motivation
- Programming with R
  - use fast instructions
- Adding Compiled Code
  - avoid interpretation for recurrent processes
- Taking full advantage of hardware
  - parallelization and GPUs
- Conclusions

Improvement in processor speed and memory capacity



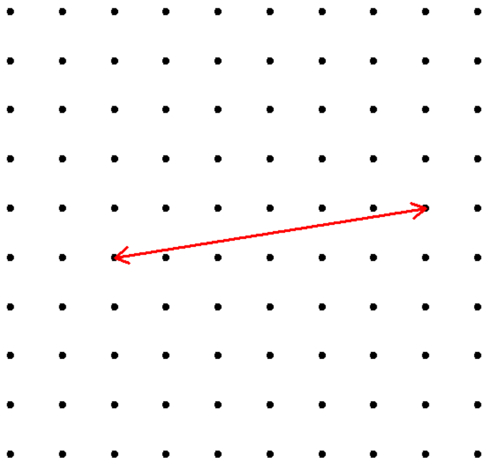
A combinatorial problem: Distances in a grid

- $n \times n$ Grid
- Two points randomly selected
- Euclidean distance between the two points
- Which is its expected value?
- Combinatorial problem: solve numerically as a function of n



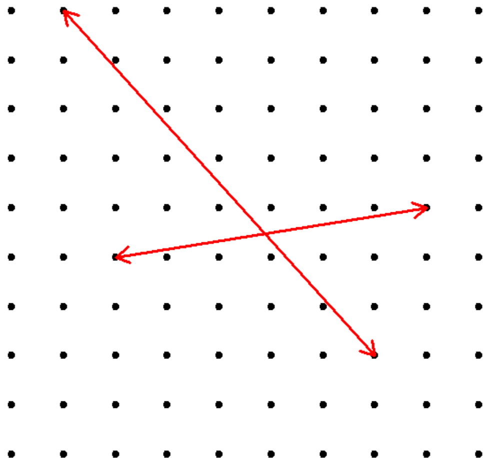
A combinatorial problem: Distances in a grid

- $n \times n$ Grid
- Two points randomly selected
- Euclidean distance between the two points
- Which is its expected value?
- Combinatorial problem: solve numerically as a function of n



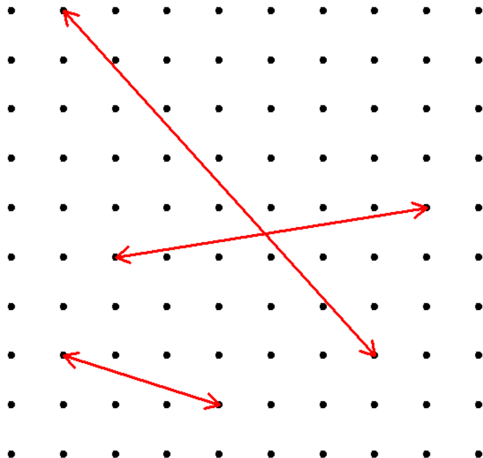
A combinatorial problem: Distances in a grid

- $n \times n$ Grid
- Two points randomly selected
- Euclidean distance between the two points
- Which is its expected value?
- Combinatorial problem: solve numerically as a function of n



A combinatorial problem: Distances in a grid

- $n \times n$ Grid
- Two points randomly selected
- Euclidean distance between the two points
- Which is its expected value?
- Combinatorial problem: solve numerically as a function of n



Distances in a grid

For each $(x_1, y_1), (x_2, y_2) \in \{1:n\} \times \{1:n\}$
calculate $d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
average all those quantities

Analytical Solution:

$$E(d) = \frac{\sum_{x_1=1}^{n} \sum_{x_2=1}^{n} \sum_{y_1=1}^{n} \sum_{y_2=1}^{n} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}}{n^4}$$

First approach with R

Four loops and dynamic memory growth

```
f1 = function(n) {
 d = NULL # No elements in d
 for (x1 in 1:n) {
  for (y1 in 1:n) {
   for (x2 in 1:n) {
    for (y2 in 1:n) { # Adding a value each step
        d = c(d, sqrt((x1 - x2)^2 + (y1 - y2)^2))
      }
     }
    }
   }
 return(mean(d))
}
```

Very slow, mainly because of the memory management

Second approach with R

Four loops and static memory management

```
f2 = function(n) {
 d = rep(0, n^4) # Initialize d with zeroes
 k = 0
 for (x1 in 1:n) {
  for (y1 in 1:n) {
   for (x2 in 1:n) {
    for (y2 in 1:n) { #Assign value to position k
        d[k <- k + 1] = sqrt((x1 - x2)^2 + (y1 - y2)^2)
      }
     }
    }
   }
 return(mean(d))
}
```

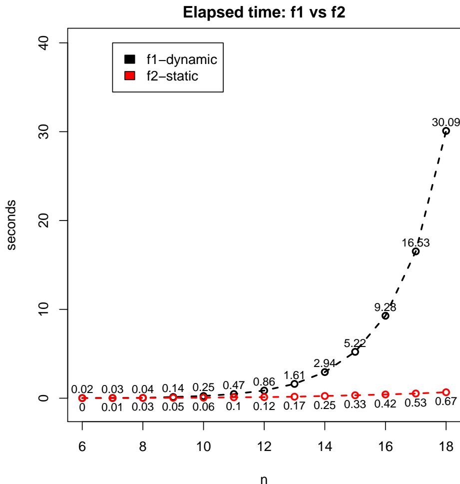
Faster, less memory fragmentation

Performance of f1 and f2

```
> system.time(print(f1(10)))

[1] 5.186872
   user  system elapsed
   0.24    0.00    0.23

> system.time(print(f2(10)))

[1] 5.186872
   user  system elapsed
   0.08    0.00    0.08
```

- user: time for user instructions of the calling process
- system: time for execution by the system
- elapsed: Total R process time (seconds)

Performance of `f1` and `f2`



It's better to reserve memory in advance!

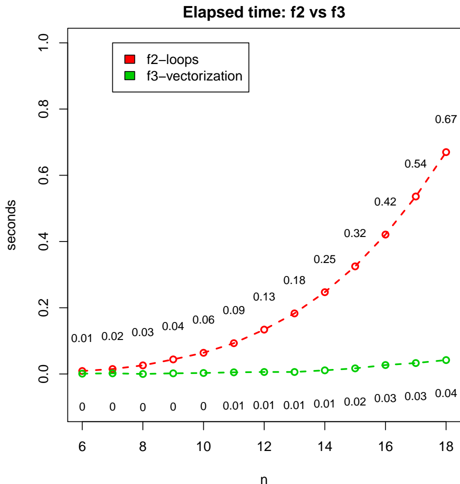
Third approach with R

Avoiding loops by using vectorization

```
f3 = function(n){
  # making all combinations!
  combi = expand.grid(1:n,1:n,1:n,1:n)

  # operations by columns
  d = sqrt((combi[,1]-combi[,3])^2+(combi[,2]-combi[,4])^2)

  return(mean(d))
}
```

Even faster, no loops!

Performance of `f2` and `f3`



Vectorial arithmetic is faster than loops!

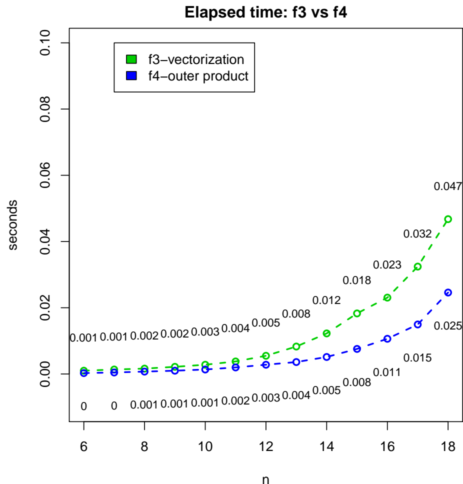
Fourth approach with R

Using specialized functions

```
f4=function(n){
    # Outer product of arrays with subtract function
    # and square
    val=outer(1:n,1:n,"-")^2
    val2=outer(1:n,1:n,"-")^2

    # Outer product of resulting arrays with add function
    # and square root
    d=sqrt(outer(val,val2,"+"))

    return(mean(d))
}
```

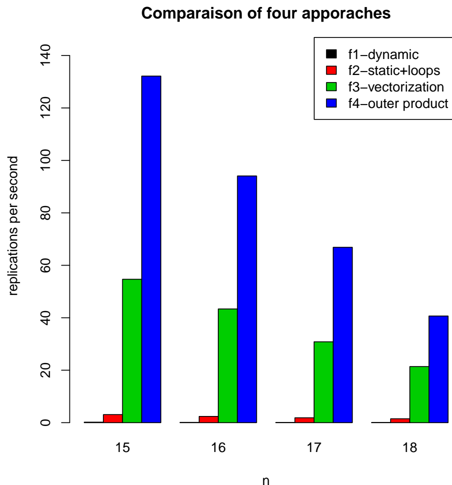
The fastest with efficient primitives (apply-like functions)!

Performance of `f3` and `f4`



Specific primitives are implemented efficiently!

Comparing all variations



There are options in R to do things faster in an easy way!

lm vs. lm.fit

Computer intensive methods

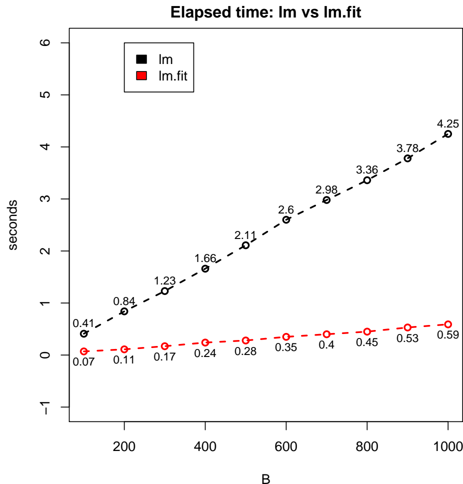
- Monte Carlo simulation
- Bootstrap
- Cross-validation
- MCMC
- ...

- Unnecessary instructions penalize computing time.
- `lm` function has an interface from a formula and generate the model matrix to solve the normal equations by calling `lm.fit`

Parametric Bootstrap: lm vs. lm.fit

```
n=100; p=5; beta=1:(p+1); s=3
X=cbind(rep(1,n),matrix(runif(p*n),nc=p))

 # linear model fit with formula interface
 lm1 = function(k){
       Y = X\%*\%beta+rnorm(n,sd=s)
       coef(lm(Y~X-1))
       }
 system.time(res1<-apply(matrix(seq(2000)),1,lm1))

 # linear model fit on matrices
 lm2 = function(k){
       Y = X\%*\%beta+rnorm(n,sd=s)
       coef(lm.fit(X,Y))
       }
 system.time(res2<-apply(matrix(seq(2000)),1,lm2))
```

Parametric Bootstrap: lm vs. lm.fit



lm.fit avoid unnecessary instructions!

Compiled Code in R

Beyond smarter code the most direct speed gain comes from switching to compiled code.

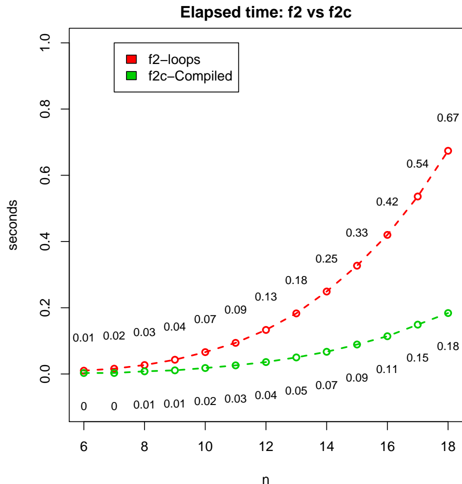
- `compiler`: Transform into byte code
- `inline`: Automated wrapping of single expression
- `Rcpp`: Interface between R and C++
- External Fortran/C/C++: Dynamic linking

Package compiler

compiler(Luke Tierney): byte code compiler for R

```
library(compiler)
f2 = function(n) {
   d = rep(0, n^4) # Initialize d with zeroes
   k = 0
   for (x1 in 1:n) {
    for (y1 in 1:n) {
     for (x2 in 1:n) {
      for (y2 in 1:n) { #Assign value to position k
          d[k <- k + 1] = sqrt((x1 - x2)^2 + (y1 - y2)^2)
        }
       }
      }
     }
   return(mean(d))
}
f2c=cmpfun(f2)
```

Performance of `f2` and `f2c`



Compiled Byte Code outperforms interpreted code!

Package inline

inline(Oleg Sklyar et al): can wrap Fortran, C or C++ code

```
library(inline)
code <- "SEXP m;
       float s = 0;
       int nt=*INTEGER(n)+1;
       for (int a = 1; a < nt; a++)
         for (int b = 1; b < nt; b++)
           for (int c = 1; c < nt; c++)
             for (int d = 1; d < nt; d++)
                s += sqrt((a-c)*(a-c)+(b-d)*(b-d));
       *REAL(m)=s/(n^4);
       return(m);"

f2i <- cfunction(sig=signature(n="integer"), body= code)
```

Rcpp / External Code

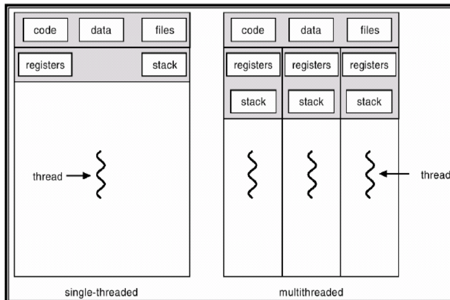
- Exposing C++ functionality to R is facilitated by the `Rcpp` package (Eddelbuettel and François)
  - Explicit constructor/destructor lifecycle of objects
  - `Rcpp` classes to avoid managing memory directly
  - Data interchange R/C++ is managed by powerful yet simple mechanisms
- With external code, it is important to coerce all the arguments to the correct R storage
  - `.C` and `.Fortran`: older and simpler
  - `.Call` and `.External`: less restrictive

Parallelization of processes

By default...

- ...R use only one core
- ...R reads data into memory

Parallelism: running several computations at the same time, taking advantage of multiple cores or CPUs



Types of parallelization

Implicit parallelism

- Set up and distributing data is done by the system.
- It is simpler but there is lack of control in the parallel features

Explicit parallelism

- User controls the cluster settings.
- It requires the user to know the specificities of the implementation

Implicit parallelization: some R Packages

- `multicore`: locally running parallel computations in R on machines with multiple cores or CPU
- `pnmath` and `pnmath0`: parallel implementation of math functions
- `fork`: wrappers around the Unix process management API calls

Only Unix versions. Not working properly on Window platforms

Explicit parallelization: Package `snow`

`snow` (Simple Network Of Workstations)

Provides an interface to several parallelization packages:

- MPI: Message Passing Interface, via Rmpi
- NWS: NetWork Spaces via nws
- PVM: Parallel Virtual Machine
- Sockets via the operating system

They allow intrasystem communication (multiple CPUs), or intersystem communication (cluster)

`snowfall`: Wrapper for `snow` with an easier interface

Package `snow`. Main functions

- `makeCluster` sets up the cluster and initializes its use
- `clusterCall` calls a specified function with identical arguments on each node in the cluster
- `clusterApply` takes a cluster, arguments and a function and calls it with the first element of the list on first node, second element on second node...
- `clusterApplyLB` same as above, with load balancing
- `parCapply`, `parRapply`, `parLapply`, `parSapply` parallel versions of column apply, row apply and the other apply variants

Parallelization: Example

Iterating the mean grid distance problem

- Repeat 20 times calculation of the mean distance in a 10×10 grid
- Record time for each iteration and total time for the whole process

```
# One thread: by default

  (t1 <- system.time(p1 <- unlist(lapply(rep(10,20),
        function(el) system.time(f2(el))["elapsed"]))))
```

Package snow and snowfall

```
# Two Threads: create a cluster with 2 sockets
 library(snow)
 cl<-makeCluster(2, "SOCK")
 clusterExport(cl,"f2")
 (t2<- system.time(p2<- parSapply(cl, rep(10,20),
    FUN=function(el) system.time(f2(el))["elapsed"])))
 stopCluster(cl)

 library(snowfall)
 sfInit(parallel=TRUE, cpus=2, type="SOCK")
 sfExport("f2")
 (t3<- system.time(p3 <- sfLapply(rep(10,20),
    function(el) system.time(f2(el))["elapsed"])))
 sfStop()
```

Performance one vs. two threads

Parcial times for each of the 20 iterations and total time

	1	2	3	4	5	6	7
One thread	0.13	0.13	0.13	0.11	0.10	0.11	0.13
Two threads	0.11	0.11	0.11	0.11	0.11	0.11	0.12

	8	9	10	11	12	13	14
One thread	0.11	0.11	0.11	0.11	0.11	0.11	0.13
Two threads	0.12	0.11	0.12	0.11	0.12	0.12	0.12

	15	16	17	18	19	20	TOTAL
One thread	0.11	0.10	0.11	0.13	0.11	0.11	**2.72**
Two threads	0.12	0.12	0.11	0.11	0.12	0.11	**1.44**

Explicit parallelization

- Sometimes, parallelization of a process can be slower than a serial approach...
- Setting up slaves, copying data and code can be very costly

General Rule

"Only parallelize with a certain method if the cost of computation is (much) greater than the cost of setting up the framework."

GPU

Computing on GPUs (Graphics Programming Units): hardware acceleration

- GPUs are hardware that is optimised for I/O and floating point operations
- Much faster code execution than standard CPUs on floating-point operations
- Development environments:
  - **Nvidia CUDA (Compute Unified Device Architecture)** Provides C-like programming
  - **OpenCL (Open Computing Language)**provides a vendor-independent interface to GPU hardware

Conclusions

- Many statistical analysis tasks are computationally very intensive
- R has enhanced its efficiency by including vectorization or specific primitives
- With computer intensive methods it is important to avoid unnecessary instructions
- It is possible to work faster with compiled code
- Even parallelization and hardware capabilities can speed up processes

Questions?

Thank you!